A PROGRAM TRANSFORMATION FROM EQUATIONAL PROGRAMS INTO LOGIC PROGRAMS(Lambda Calculus and Computer Science Theory)

AUTHOR(S):

TOGASHI, Atsushi; NOGUCHI, Shoichi

数理解析研究所講究録
515巻 1984年 62-88

A PROGRAM TRANSFORMATION FROM EQUATIONAL PROGRAMS INTO LOGIC PROGRAMS

Atsushi TOGASHI, Shoich NOGUCHI
RESEARCH INSTITUTE OF ELECTRICAL COMMUNICATION,
TOHOKU UNIVERSITY, SENDAI in JAPAN

ABSTRACT In this paper we propose a transformation algorithm from equational programs written in equational form into logic programs. In order to facilitate program transformation we extend the programming language Prolog into a new logic programming language based on a new computation model called a cluster reduction system. It is shown that any equational program is transformed into an equal or more powerful logic program. As for a recursive equational program, there exists a logic program with the equivalent computational power.

We believe our paper is the first attempt to clarity relationship among descriptive languages. Our results suggest the introduction of the notion data abstraction and computation strategies into a logic programming language.

1. Introduction

In the last years substantial efforts have been made to develop an equational programming language[2,4,8,14] and / or a logic programming language[1,5,9,12], so called descriptive languages. Both languages are based on some mathematical systems and show certain similarities each other. This indicates some possibility of program transformation. As for program transformation, the equational language concerns with an algebraic specification for abstract data types[6] and a recursive program scheme [2,4]. The transformation provides introduction of notions data abstraction and computation strategies in a logic programming language.

In this paper we propose a transformation algorithm from equational programs into logic programs. It is shown any equational program is transformed into an equal or more powerful logic program. When we restrict our attention to recursive equational programs any recursive equational program is simulated by some Horn program with the equivalent computational power.

In order to facilitate program transformation we extend the programming language Prolog, which was more popular and investigated by many researchers, into a new logic programming language based on a new computational model. This language is more suitable for representation of knowledge for predicates, and is oriented to knowledge based programming.

This paper is organized as follows : In chapter 2 some preliminary definition are discussed. The formulation of equational and logic programming languages are described in chapter 3 and in chapter 4,

respectively. There some results are also considered. In chapter 5 we propose a transformation algorithm and validity of the method is proved.

2. Preliminary Definitions

In this chapter some preliminary definitions and basic results for them are presented. In particular, we state some notions signatures, terms, substitutions, and term rewriting systems. See (10,11,14,15) for detail discussions.

2.1 Signatures, Terms and Substitutions

It is assumed that we are given a finite set S of "sorts"[11], which are names of various data types[6] under consideration.

Definition 1 A (S-sorted) "signature" is an indexed family $\{\sum_{w,s}\}_{(w,s)\in S^* \times S}$ of disjoint sets $\sum_{w,s}$, where S^* denotes a set of all finite sequences on S with a null string Λ (S^+ is a set of all non null sequences on S). A symbol $\sigma \in \sum_{w,s}$ is called a "function symbol" of sort s with arity w, sometimes written $\sigma : w \rightarrow s$. If $w = \Lambda$, σ is called a "constant". (+)

For ease of notation, let $\sum = \bigcup_{(w,s)\in S^* \times S} \sum_{w,s}$, and we use $\sum$ to denote the signature.

Let $X = \bigcup_{s \in S} X_s$ be a disjoint union of denumerable sets X_s of "variables" of sorts s and fixed throughout this paper.

Definition 2 For a signature $\sum$, "$\sum$-terms" (or "terms" whenever $\sum$ is clear from the context) t of sorts s together with sets Var(t) of variables appearing in t are defined in the recursive way:

(1) Each variable $x \in X_s$ is a $\sum$-term of sort s, and $Var(x) = \{x\}$;

(2) Each constant $\sigma : \Lambda \rightarrow s$ is a $\sum$-term of sort s, and $Var(\sigma) = \emptyset$;

(3) If $\sigma : s_1, \ldots, s_n \to s$ is a function symbol and t_i are Σ-terms of sorts s_i, then $t = \sigma(t_1, \ldots, t_n)$ is a Σ-term of sort s, and

$$Var(t) = Var(t_1) \cup \ldots \cup Var(t_n). \quad (+)$$

The set of all Σ-terms of sort s is denoted by $T(\Sigma,X)_s$. Finally we define $T(\Sigma,X)$ as a disjoint union of sets $T(\Sigma,X)_s$, $s \in S$.

We now formalize notions of subterms and occurrences of subterms in the term. Let N^* be a set of all strings on the set N of positive integers with a null string λ. We shall call the members of N^* "occurrences" and denote them u, v and w, possibly with subscripts.

Definition 3 Given a Σ-term t in $T(\Sigma,X)$ we define its set of occurrences $Ocr(t) \subset N^*$ and a "subterm" t/u of t at the occurrence $u \in Ocr(t)$ as follows:

(1) If t is a variable or a constant, then

$$Ocr(t) = \{\lambda\}; \quad t/\lambda = t.$$

(2) If t is of the form $\sigma(t_1, \ldots, t_n)$ for some function symbol $\sigma : s_1, \ldots, s_n \to s$, then

$$Ocr(t) = \{\lambda\} \cup \{iu \mid 1 \le i \le n, \ u \in Ocr(t_i)\};$$

$$t/\lambda = t \quad \text{and} \quad t/iu = t_i/u. \quad (+)$$

We say u is an occurrence of the subterm t/u in t.

Definition 4 For terms t, t' and $u \in Ocr(t)$, we define $t[u \leftarrow t']$ as the term t, in which the subterm t/u at the occurrence u is replaced by t'. (+)

Definition 5 A "substitution" is a mapping $\theta : X \to T(\Sigma,X)$ such that $\theta(x) = x$ almost everywhere, that is the domain of θ defined by $Dom(\theta) =$

$\{ x : \theta(x) \neq x \}$ is finite. Here we impose that all variables of sort s are mapped into terms of sort s by θ. (+)

The substitution θ is extended into terms by

$$\theta(\sigma(t_1, \ldots, t_n)) = \sigma(\theta(t_1), \ldots, \theta(t_n))$$

where $\sigma : s_1, \ldots, s_n \rightarrow s$ is a function symbol and t_i are terms of sorts s_i.

Definition 6 we define the quasi-ordering $\lhd$ (the reflexive and transitive relation) on terms by

$$t \lhd t' \text{ if and only if } t' = \theta(t) \text{ for some substitution } \theta$$

for all terms t, t' in $T(\Sigma, X)$. (+)

2.2 Term Rewriting Systems

Definition 7 A "term rewriting system" on a signature Σ is a finite set R of "rewriting rules" of the form $l \rightarrow r$ such that $Var(l) \supset Var(r)$, where l and r are Σ-terms of the same sorts. (+)

R may be "applicable" to a term t if and only if there is an occurrence $u \in Ocr(t)$, called a "redex occurrence"[11] of R in t, such that $l \lhd t/u$ for some rewriting rule $l \rightarrow r$ in R. In this case, we say that the rule $l \rightarrow r$ is applied to the term t to obtain the term $t[u \leftarrow \theta(r)]$, where θ is the unique substitution such that $t/u = \theta(l)$. The choice of which rules to apply is made non deterministic. We write $t \Rightarrow_R t'$ to indicate that the term t' is obtained from the term t by a single application of some rule in R. Let $\overset{*}{\Rightarrow}_R$ denote the reflexive, transitive closure of $\Rightarrow_R$. If $t \overset{*}{\Rightarrow}_R t'$ holds we say t' is "derivable" from t in R. R may be omitted from $\overset{*}{\Rightarrow}_R$

and $\Rightarrow_R$ when it is clear from the context. The derivation relation is characterized in the proof system in the following way.

Proposition 1 Let R be a term rewriting system on Σ and t, t' be any Σ-terms. Then $t \overset{*}{\Rightarrow} t'$ holds if and only if an ordered pair of terms $t \geq t'$ is provable in the proof system with the following inference rules.

(1) $$\frac{l \to r \quad R}{l \geq r}$$

(2) $$\frac{}{t \geq t}$$

(3) $$\frac{t \geq t', \quad t' \geq t''}{t \geq t''}$$

(4) $$\frac{t_i \geq t_i', \quad \sigma : s_1, \ldots, s_n \to s}{\sigma(t_1, \ldots, t_n) \geq \sigma(t_1', \ldots, t_n')}$$

(5) $$\frac{t \geq t', \quad \theta : X \to T(\Sigma, X)}{t\theta \geq t'\theta}$$

proof. Both directions can be easily verified by induction, so we omit the proof. (+)

remark The notation $t \geq t'$ of ordered pairs comes from the fact that ordered pairs provable in the proof system are characterized by the partial ordering relation on terms. See (10,11).

3. Equational Programs

In this chapter, we formulate an equational program in the framework of a term rewriting system[8,14].

Let Σ be a (finite) S-sorted signature. The signature Σ is partitioned as $\Sigma = \Sigma^c \cup \Sigma^d$. We call function symbols in Σ^c constructors, and members in Σ^d defined function symbols. We assume that there is at least one constructor for each sort s in S.

Definition 8 An "equational program" on the signature Σ is a term rewriting system R in which each rewriting rule is of the form

$$F(E_1, \ldots, E_n) \dashrightarrow E_{n+1},$$

where F is a defined function symbol and E_i are Σ-terms. (+)

Constructors create data types. Defined function symbols define some manipulations over the constructed data types; the meaning of them are described by rewriting rules. For notational convenience, constructors are denoted by lower case letters, and defined function symbols by capital letters such as F, G, H and so on. Similarly we use symbols E, E_i to denote Σ-terms and t, t_i to denote Σ^c-terms, called "constructor terms", constructed only by constructors. Of course, both kinds of terms contain variables as constituents.

A certain restriction can be made on the nature of the rewriting rules to give more restricted class of equational programs.

Definition 9 An equational program R is "recursive" if every rule in R is of the form

$$F(t_1, \ldots, t_n) \dashrightarrow E,$$

where t_i are constructor terms. (+)

Recursive equational program can be viewed as generalization of non deterministic recursive program schemes[2].

Let R be an equational program. A "computation (sequence)" of an input term (expression) E_0 is a possibly infinite derivation sequence

$$E_0 \Rightarrow_R E_1 \Rightarrow_R \cdots$$

The computation of the input E_0 "successfully terminates" if E_n is a constructor term t for some $n \geq 0$, hence we can not rewrite E_n any more by definition of equational programs. In this case, $E_n = t$ is a "result" of this computation. Otherwise the computation "fails", that is it terminates at the term E which contains some defined function symbols or never terminates.

Example 1 An equational program R reversing lists

constructors :

nil : Λ --> list;

cons : item, list --> list;

defined function symbols :

APPEND : list, list --> list;

REV : list --> list

rewriting rules :

APPEND(nil, x) --> x

APPEND(cons(i,x), y) --> cons(i, APPEND(x,y))

REV(nil) --> nil

REV(cons(i,x)) --> APPEND(REV(x), cons(i,nil))

REV(REV(x)) --> x

APPEND(APPEND(x,y), z) --> APPEND(x, APPEND(y,z))

REV(APPEND(x,y)) --> APPEND(REV(y), REV(x))

(+)

What we have defined above is the most general strategy of executing programs. A more restricted strategy is treated here to be simulated by logic programs.

Definition 10 Let R an equational program. A $\sum$-term E' is derivable from a $\sum$-term E in a "primitive execution strategy" denoted by $E \underset{(p)}{\Rightarrow} E'$ if there exist a rule $l \rightarrow r$, an occurrence u Ocr(E) and substitution $\theta : X \rightarrow T(\sum^C, X)$ with the range the set of only constructor terms such that $E/u = \theta(l)$ and $E[u \leftarrow \theta(r)] = E'$. (+)

A computation from E_0 in a primitive execution strategy and result for it are defined similarly to the case of general strategy.

Corollary 1 Let R be an equational program and E, E' be $\sum$-terms. $E \underset{(p)}{\overset{*}{\Rightarrow}}_R E'$ holds if and only if $E \geq E'$ is provable by applying the inference rules (1), (2), (3), (4) mentioned in Proposition 1 and

(5') $$\frac{E \geq E', \quad \theta : X \rightarrow T(\sum^C, X)}{E\theta \_ E'\theta}$$

(+)

4. Logic Programs.

In the last few years a programming language Prolog[1,5,12] based on the Horn clauses[7] in the first order logic has been increasingly used, due to the possibility of suitably using it as a specification language and as a practical, efficient programming language.

In order to facilitate transformation from equational programs into logic programs, we extend Prolog into a logic programming language to have more than one atoms in their left-hand sides of Horn clauses. Also we

introduce inferred variables which will be distinguished from fixed variables. As an example, the left distributive law of the multiplication MULT for the addition ADD can be expressed as

$$\text{MULT}(x,y_1,*u),\ \text{MULT}(x,y_2,*v),\ \text{ADD}(*u,*v,z)$$
$$:-\ \text{MULT}(x,*w,z),\ \text{ADD}(y_1,y_2,*w).$$

This corresponds to the usual distributive law of the multiplication "." over the addition "+" written by the equation

$$x \cdot y_1 + x \cdot y_2 = x \cdot (y_1 + y_2)$$

This formula has more than one atoms at the left-hand side and variables appearing in it are partitioned into two kinds of variables. One is a fixed variable such as x or z, bounded by universal quantifier $\forall$ from outside, the other is an inferred variable such as *u or *v, bounded by existential quantifier $\exists$ from inside. The above formula can be expressed by the usual form such as

$$\forall x, \forall y_1, \forall y_2, \forall z :$$
$$\exists u, \exists v : \text{MULT}(x,y_1,u),\ \text{MULT}(x,y_2,v),\ \text{ADD}(u,v,z)$$
$$<\!\!-\!\!-\ \exists w : \text{MULT}(x,w,z),\ \text{ADD}(y_1,y_2,w)$$

This formula asserts a single concept, hence can not be modified into more than one definite clauses[1] without loosing the flavor it has. This kind of property can be used to simplify subgoals and speed up its computation. The application of the above property is allowed only subgoals in which there are some atoms identified with its whole left hand side of the rule by two kinds of substitution, and results in a parallel rewriting of atoms. In this way, we have modified the way subgoals are

computed according to the above extension, so that properties will be applied to get an intelligent, efficient computation.

Definition 11 A (S-sorted) "similarity type" is a pair $d = (\sum^c, \Pi)$, where $\sum^c$ is a S-sorted signature and Π is a disjoint union of sets Π_w of predicate symbols P, written by $P : w$, for $w \in S^+$. (+)

Definition 12 Let $d = (\sum^c, \Pi)$ be a S-sorted similarity type.

(1) An "atomic formula" (or "atom", in short) is $P(t_1, \ldots, t_n)$, where $P : s_1, \ldots, s_n$ is a predicate symbol and t_i are terms of sorts s_i.

(2) A "cluster formula" (or "cluster") is a finite set of atomic formulas. (+)

For a cluster M, Var(M) denotes a set of all variables appearing in M. There are two kinds of variables, that is "fixed variable" and "inferred variables", which correspond the variables bounded by universal quantifier $\forall$ and by existential quantifier $\exists$, respectively. We assume that the set Var(M) is partitioned into two sets, that is a set FIX(M) of fixed variables and a set INF(M) of inferred variables.

For simplicity convention, we will write a cluster $C_1, \ldots, C_k$ rather than $\{C_1, \ldots, C_k\}$. For this reason, the order of atoms in the cluster is not so crucial. If $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ are fixed variables and inferred variables of the cluster $M = C_1, \ldots, C_k$, we can read it as

for all $x_1, \ldots, x_m$ there exist $y_1, \ldots, y_n$ such that

$$C_1 \text{ and } \ldots \text{ and } C_k.$$

Definition 13 Let d be a similarity type. A "cluster sequent" on d is an ordered pair of clusters of the form M :- N which satisfy the following two conditions:

(a) All fixed variables appearing in the right hand side also appear in the left hand side, i.e., $FIX(M) \supset FIX(N)$;

(b) there is no common variable among FIX(N), INF(M) and INF(N).

The cluster M is called a "conclusion" of the cluster sequent; the cluster N is called a "premise" of the sequent. (+)

For a cluster $r = A_1, \ldots, A_m :- B_1, \ldots, B_n$, let $FIX(A_1, \ldots, A_m) = \{ x_1, \ldots, x_k \}$, $INF(A_1, \ldots, A_m) = \{ y_1, \ldots, y_p \}$, and $INF(B_1, \ldots, B_n) = \{ z_1, \ldots, z_q \}$. The cluster sequent $r = A_1, \ldots, A_m :- B_1, \ldots, B_n$ can be interpreted as for all $x_1, \ldots, x_k$:

if there exist $z_1, \ldots, z_q$ such that B_1 and ... and B_n, we can assert the existence of $y_1, \ldots, y_p$ such that A_1 and ... and A_m.

Definition 14 A "definite sequent" is a cluster sequent of the form $A :- B_1, \ldots, B_n$ such that $INF(A) = \emptyset$. (+)

Definite sequents correspond to definite clauses. The alternative formulation derives from the fact a universally quantified implication

$$(\forall x) : A <-- B_1, \ldots, B_n$$

is logically equivalent to

$$A <-- (\exists x) : B_1, \ldots, B_n$$

when x does not occur in A.

Definition 15 A "logic program" on a similarity type d is a finite set £ of cluster sequents. If £ consists only of definite sequents, £ is said to be a "Horn program". (+)

A "goal" for a logic program is a cluster. Goals describe some problems which will be solved by the execution of programs. In the procedure interpretation a logic program is a goal reduction (replacement) system

likewise a problem reduction system[16]. A computation (or an execution) of programs is initiated by giving an input goal. The computation proceeds by apply some cluster sequents to derive successive new subgoals. In each computation step some subcluster is selected from the subgoal and matched with the left hand side of some cluster sequent by finding two kinds of appropriate substitutions. The subcluster is then replaced by the right hand side of the cluster sequent. The program successfully terminates for the input goal if the empty goal (terminal goal) is derived.

In the following, we will formalize a way goals are computed according to the extention mentioned before.

Definition 16 Let £ be a logic program. A cluster sequent $r = A_1, \ldots, A_m :- B_1, \ldots, B_n$ in £ is "applicable" to a goal $M = C_1, \ldots, C_k$ iff there exist two substitution θ, η with conditions

$$Dom(\theta) \subset FIX(A_1, \ldots, A_m), \quad Dom(\eta) \subset INF(C_1, \ldots, C_k),$$

called "matching substitution" and "inferring substitution", respectively such that

$$(C_1, \ldots, C_m)\eta = (A_1, \ldots, A_m)\theta$$

for some subcluster $C_1, \ldots, C_m$ of $C_1, \ldots, C_m, C_{m+1}, \ldots, C_k$. (+)

If r in £ is applicable as above, we say the sequent r is applied to obtain a new goal

$$N = (B_1, \ldots, B_n)\theta, (C_{m+1}, \ldots, C_k)\eta.$$

Inferred variables of N are defined by

$$INF(N) = \{ x \in INF(M) : x \notin Dom(\eta) \} \cup INF(B_1, \ldots, B_n)$$
$$\cup \{ x \in Var(A_1, \ldots, A_m) : x \notin Dom(\theta) \} .$$

All others variables appearing in N are fixed variables.

Example 2 Let us consider a logic program consisting of a single cluster sequent

$$A(x,z,\ f(x,*v))\ \div\!\!-\ \ A(g(z),\ *w,x).$$

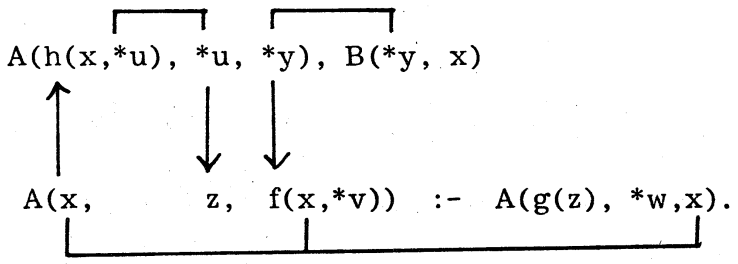
In order to distinguish inferred variables with fixed variables we use a symbol "$_*$" in such a way *x stands for that x is an inferred variable. Given a goal

$$A(h(x,*u),\ *u,\ *y),\ B(*y,\ x)$$

£ is applicable to it, and we obtain a new goal

$$A(g(*z),\ *w,\ h(x,*z)),\ B(f(h(x,*z),\ *w),\ x)$$

as a result of application.



$$A(g(*z),\ *w,\ h(x,*z)),\ B(f(h(x,*z),\ *w),\ x)$$

Fig.1 An application of a cluster sequent

(+)

For goals M, N $M \Rightarrow_{£} N$ indicates that N is obtained from M by a single application of some sequent in £. We may write $M \Rightarrow^{\eta}_{£} N$ to clarity the used inferring substitution η . $\overset{*}{\Rightarrow}_{£}$ denotes reflexive, transitive closure of $\Rightarrow_{£}$. If $M \overset{*}{\Rightarrow}^{\eta}_{£} N$ holds, we say M is "reducible" to N, where η is a composition of used inferring substitutions.

Proposition 2 Let £ be a logic program and M, N be goals. For any substitution ζ with the domain contained by INF(M), we have $M\zeta \Rightarrow_{£} \eta$ N implies $M \Rightarrow_{£} \zeta\eta$ N. (+)

Corollary 2 In the same condition as Proposition 2 it follows that $M\zeta \overset{+}{\Rightarrow}_{£} \eta$ N implies $M \overset{+}{\Rightarrow}_{£} \zeta\eta$ N, where $\overset{+}{\Rightarrow}_{£}$ is a transitive closure of $\Rightarrow_{£}$. (+)

Conversely, we can easily verify the following Proposition by induction on the length of reductions.

Proposition 3 Let £ be a logic program. $M\eta \overset{*}{\Rightarrow}_{£}$ N follows from the condition $M \overset{*}{\Rightarrow}_{£} \eta$ N for all goals M, N. (+)

Let £ be a logic program. A "computation" from a goal is a reduction sequence

$$M = M_0 \Rightarrow_{£} \eta_1 \; M_1 \Rightarrow_{£} \eta_2 \; \ldots \; .$$

A computation "successfully terminates" if M_n is an empty goal, denoted by e, for some $n \geq 0$, where an empty goal is an empty cluster. In this case, a composition $\eta = \eta_1 \ldots \eta_n$ is an "answer substitution" and $M\eta$ is a "result" for the computation.

5. A Transformation Algorithm.

At first, we show a method for transforming a given Σ-term E into a cluster C(E) and an output term O(E) associated with E.

Let $\Sigma = \Sigma^c \cup \Sigma^d$ be a S-sorted signature for equational programs. A S-sorted similarity type $d = (\Sigma^c, \Gamma)$ for logic programs is specified in terms of Σ in the following way:

(a) A set Σ^c of function symbols is identical to constructors in Σ;

(b) A set Γ of predicate symbols is defined by

$$\Gamma = \{ F_P : s_1, \ldots, s_n, s \mid F : s_1, \ldots, s_n \rightarrow s \in \Sigma^d \}.$$

Algorithm A

We associate a cluster C(E) and an output term O(E) with a Σ-term E by structural induction on E. For a cluster C(E) fixed variables are ones which belong to Var(E) and newly introduced variables are inferred variables.

(1) If E is a variable $x \in X$ or constant $a : \Delta \rightarrow s \in \Sigma^c$, define

$$C(E) = e; \quad O(E) = E,$$

where e is a empty cluster.

(2) Suppose E is of the form

$$E = \sigma(E_1, \ldots, E_n),$$

where $\sigma : s_1, \ldots, s_n \rightarrow s \in \Sigma$ is a function symbol. (By induction hypothesis, it is assumed that clusters $C(E_i)$ and output terms $O(E_i)$ are constructed in such a way that newly introduced variables are standardized apart one another.)

(a) in case σ is a constructor f, define

$$C(E) = C(E_1), \ldots, C(E_n);$$

$$O(E) = f(O(E_1), \ldots, O(E_n)).$$

(b) in case σ is a defined function symbol F, define

$$C(E) = C(E_1), \ldots, C(E_h), F_P(O(E_1), \ldots, O(E_n), *y)$$

$$O(E) = *y.$$

where F_P is the predicate symbol corresponding to F and *y is the new variable which never appears in $C(E_i)$ for all $1 \leq i \leq n$. (+)

Example 3 Let us consider a Σ-term

$$E = f(F(G(x), g(a)), H(G(x))),$$

where f, g, a are constructors (a is a constant) and F, G, H are defined symbols with arbitrary types. By applying Algorithm A to this $\sum$-term we obtain a cluster C(E) and an output term below:

$$C(E) = G_P(x,*y_3),\ F_P(*y_3,g(a),*y_1),\ G_P(x,*y_4),\ H_P(*y_4,*y_2)$$

$$O(E) = f(*y_1,\ *y_2).$$

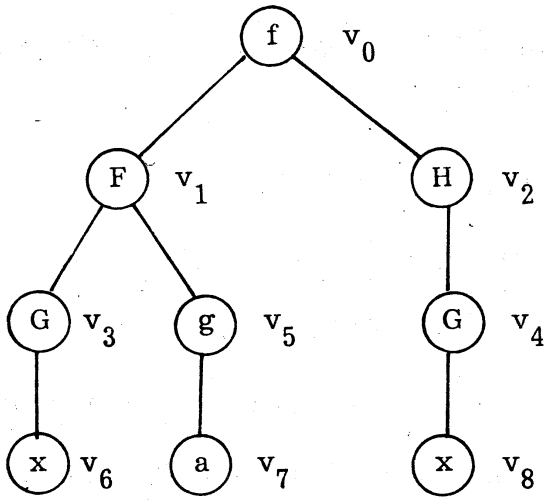


Fig.2 A tree representing a $\sum$-term E.

By using Algorithm A a transformation algorithm from equational programs into logic programs is described in the following way:

Algorithm B

Let R be a given equational program. We translate each rewriting rule $F(E_1, \ldots, E_n) \dashrightarrow E'$ in R into a cluster sequent to construct a corresponding logic program.

(i) Constitute clusters and output terms of both sides of rule by using Algorithm A. During execution of algorithm newly defined variables are

standardized apart in both sides. (Let note that the output term of left hand side must be the variable.)

(ii) The transformed cluster sequent is defined by

$$C(E_1), \ldots, C(E_n), F_P(O(E_1), \ldots, O(E_n), O(E')) :- C(E').$$

Fixed variables are specified by

$$Var(F(E_1, \ldots, E_n)) \cup Var(O(E'))$$

and all others are inferred variables. (+)

Example 4 If we apply Algorithm B to the equational program in Example 1, we obtain the following translated logic program.

```
APPEND(nil, x, x) :- e
APPEND(cons(i,x), y, cons(i,z)) :- APPEND(x,y,z)
APPEND(x,y,*u), APPEND(*u,z,w)
        :- APPEND(y,z,*v), APPEND(x,*v,w)
REV(nil, nil) :- e
REV(cons(i,x), z) :- REV(x, *y), APPEND(*y, cons(i,nil),z)
REV(x,*y), REV(*y,x) :- e
APPEND(x,y,*w), REV(*w,z)
        :- REV(y,*v), REV(x,*u), APPEND(*v,*u,z)          (+)
```

Proposition 4 Let R be an equational program and £ be a transformed logic program. If R is recursive, £ is the Horn program. (+)

To investigate relationship between equational programs and translated logic programs, we shall consider clusters and output terms associated with $\sum$-terms by Algorithm A. In Algorithm A we construct a cluster C(E) and output term O(E) for $\sum$-term E. Similar to clusters, we partitioned a set Var(O(E)) of variables into two kinds of variables, that is fixed variables

and inferred variables. Here fixed variables are ones which belong to Var(E).

Definition 17 For constructor terms t, t' with variables partitioned into fixed and inferred variables t is a "variant" of t' if t differs from t' at most in the names of its inferred variables. (+)

A variant of a cluster is defined similar to the variant of the constructor term.

Theorem 1 Let R be an equational program and £ denote a transformed logic program from R. For any Σ-terms E, E' if $E \overset{*}{\underset{(p)}{\Rightarrow}}_R E'$ (in the equational program R), then there exist some variants M' and t' of C(E') and O(E'), respectively, such that $C(E) \overset{*}{\Rightarrow}_{£} \eta\ M'$ (in the logic program £) and $O(E)\eta = t'$ for some inferring substitution η.

proof. By Corollary 1 the proof consists of examining each of rules of inference. As for rules of inference (1), (2) and (3) the assertion of theorem is obviously clear from the transformation algorithm. So we discuss only inference rules (4), (5').

for the inference rule (4) :

Suppose that the given terms are of the form

$$E = \sigma(E_1, \ldots, E_n)$$

$$E' = \sigma(E_1', \ldots, E_n')$$

for some function symbol σ. By structural induction we assume that

$$C(E_i) \overset{*}{\Rightarrow}_{£} \eta_i\ M_i', \quad O(E_i)\eta_i = t_i'$$

for some variants M_i', t_i' of $C(E_i')$, $O(E_i')$, and for some inferring substitutions η_i for all $1 \leq i \leq n$.

Without loss of generality we can impose conditions on clusters and output terms in such a way that

$$INF(M_i) \cap INF(M_j) = \emptyset \ ;$$
$$INF(t_i) \cap INF(t_j) = \emptyset$$

and

$$INF(M_i) \cap INF(M_j') = \emptyset \ ;$$
$$INF(t_i) \cap INF(t_j') = \emptyset$$

for all $i \neq j$, where $M_i = C(E_i)$, and $t_i = O(E_i)$. There are two possibilities for $\cap$ as the function symbol.

(a) If $\cap$ is a constructor f, then the associated clusters C(E), C(E') and output terms O(E), O(E') must be of the form

$$C(E) = C(E_1), \ \ldots \ , C(E_n);$$
$$C(E') = C(E_1'), \ \ldots \ , C(E_n'),$$

and

$$O(E) = f(O(E_1), \ \ldots \ , O(E_n));$$
$$O(E') = f(O(E_1'), \ \ldots \ , O(E_n')),$$

respectively.

Let define an inferring substitution η as a composition

$$\eta = \eta_1 \cdots \eta_n.$$

By assumption described above we have

$$C(E) = C(E_1), \ \ldots \ , C(E_n)$$
$$\overset{*}{\Rightarrow}_{\mathfrak{L}} \eta_1 \ \ M_1', C(E_2), \ \ldots \ , C(E_n)$$
$$\ldots$$
$$\overset{*}{\Rightarrow}_{\mathfrak{L}} \eta_n \ \ M_1', M_2', \ \ldots \ , M_n',$$

and

$$O(E) = f(O(E_1), \ldots, O(E_n))\eta_1 \ldots \eta_n$$
$$= f(O(E_1)\eta_1, O(E_2), \ldots, O(E_n))\eta_2 \ldots \eta_n$$
$$= f(t_1', O(E_2), \ldots, O(E_n))\eta_2 \ldots \eta_n$$
$$\ldots$$
$$= f(t_1', \ldots, t_n')$$

which are variants of C(E') and O(E'), respectively.

(b) If $\wedge$ is a defined function symbol F, then associated clusters and output terms are of the form

$$C(E) = C(E_1), \ldots, C(E_n), F_p(O(E_1), \ldots, O(E_n), *y)$$
$$C(E') = C(E_1'), \ldots, C(E_n'), F_p(O(E_1'), \ldots, O(E_n'), *y')$$

and

$$O(E) = *y, \quad O(E') = *y'$$

where F_p is a predicate symbol corresponding to the defined function symbol F and *y, *y' are new inferred variables. Similar to case (a), we can easily verify that

$$C(E) \overset{*}{\Rightarrow}_{\mathfrak{L}} \eta\ M' \quad \text{and} \quad O(E)\eta = t'$$

for some variants M', t' of C(E'), O(E'), respectively, and for some inferring substitution η, so we omit proof of them.

for inference rule (5') :

Finally for given Σ-terms E, E' suppose that

$$C(E) \Rightarrow_{\mathfrak{L}} \eta_1\ M_1 \Rightarrow_{\mathfrak{L}} \ldots \Rightarrow_{\mathfrak{L}} M_{k-1} \Rightarrow_{\mathfrak{L}} \eta_k\ M',$$

and

$$O(E)\eta_1 \ldots \eta_k = t'$$

for some variants M', t' of C(E'), O(E'), respectively. Let $\theta : X \to T(\Sigma^c, X)$ be any substitution with the range the set of constructor terms.

Define inferring substitutions $\zeta_i = \eta_i\theta$, $1 \leq i \leq n$, which map inferred variables $*y \in Dom(\eta_i)$ to terms $(*y)\eta_i\theta$. It is obviously clear by assumption that

$$C(E\theta) \Rightarrow_{\mathfrak{L}} \zeta_1 \Rightarrow_{\mathfrak{L}} \ldots \Rightarrow_{\mathfrak{L}} \zeta_k \; M'\theta$$

$$O(E\theta)\zeta_1 \ldots \zeta_k = E'\theta.$$

Hence the proof is complete. (+)

Corollary 3 Let E be a Σ-term and t a constructor term. If $E \overset{*}{\underset{(P)}{\Rightarrow}}_R t$ in the equational program R, then $C(E) \overset{*}{\Rightarrow}_{\mathfrak{L}} \eta$ e and $O(E)\eta = t$ in the corresponding logic program £ for some inferring substitution η. (+)

Without loss of generality we can assume that input terms in equational programs are of the form $F(t_1, \ldots, t_n)$, where t_i are constructor terms.

Corollary 4 If $F(t_1, \ldots, t_n) \overset{*}{\Rightarrow}_R t$ in an equational program R, then $F_P(t_1, \ldots, t_n, t) \overset{*}{\Rightarrow}_{\mathfrak{L}} e$ in the corresponding logic program for all Σ-term E and for all constructor terms t.

proof. By Proposition 3 and Corollary 3. (+)

These results indicate that in a primitive execution strategy any equational program is transformed into a equal or more powerful logic program. On the other hand, for a recursive equational program, we can construct a logic program with the equivalent computation power.

Let W denote a set of all atoms (containing variables) on a similarity type d. With a Horn program £ we associate a mapping $T_{\mathfrak{L}}$ over the power set 2^W of W.

Definition 18 Given a Horn program £, a mapping $T_{\mathfrak{L}}$ over 2^W associated with £ is defined as follows :

For any subset $V \subset W$ and for any definite sequent in £

8.

$$B_0 :- B_1, \ldots, B_n,$$

if there exists a substitution θ such that $B_i\theta \in V$ for all i, $1 \leq i \leq n$, then we have $B_0\theta \in T_{\mathcal{L}}(V)$. (+)

By definition $T_{\mathcal{L}}$ is the continuous mapping over 2^W with partial order set-theoretic inclusion among subsets on W. So $T_{\mathcal{L}}$ has a unique fixed point lfp($\mathcal{L}$) like as the result by Emden and Kowalski [5]. In fact, lfp($\mathcal{L}$) turns out to be lfp($\mathcal{L}$) = $U_{k \geq 0}\ T_{\mathcal{L}}^{k}(\emptyset)$, where $\emptyset$ is the empty subset of W.

Theorem 2 Let $\mathcal{L}$ be a Horn program and M a goal. If there is a successfully terminating computation from M with an answer substitution η, then $A\eta \in$ lfp($\mathcal{L}$) for every atomic cluster A in M.

proof. Let $M_0 \Rightarrow_{\mathcal{L}} \eta 1\ M_1 \Rightarrow_{\mathcal{L}} \ldots \Rightarrow_{\mathcal{L}} \eta k\ M_k$ be a successful computation from M with an answer substitution η. Note that $M_0 = M$, $M_k = e$ and $\eta = \eta_1 \ldots \eta_k$. We show by induction on $i \geq 1$ that

$$A\eta_{k-i+1} \cdots \eta_k \in T_{\mathcal{L}}^{i}(\emptyset)$$

for any atomic cluster A in M_{k-i},

If $i = 1$, then M_{k-1} consists of a single atomic cluster, say A. By assumption it follows that

$$A\eta_{k-1} = B_0\theta$$

for some definite sequent B_0 :- in $\mathcal{L}$ and for some matching substitution θ. Hence $A\eta_{k-1} \in T_{\mathcal{L}}(\emptyset)$ by definition of $T_{\mathcal{L}}$. This is the induction basis.

Let $i \geq 1$. Suppose that $A\eta_{k-i+1} \ldots \eta_k \in T_{\mathcal{L}}^{i}(\emptyset)$ for any atomic cluster A in M_{k-i}. Let

$$M_{k-i-1} = C_1, \ldots, C_j, \ldots, C_m$$

$$M_{k-1} = (C_1, \ldots, C_{j-1})\eta_{k-i},$$

$$(B_1, \ldots, B_n)\theta, (C_{j+1}, \ldots, C_m)\eta_{k-i}$$

for some definite sequent B_0 :- B_1, ... , B_n in £ and for some matching substitution θ, where

$$C_j \eta_{k-i} = B_0\theta$$

holds. Let A be any atomic cluster in M_{k-i-1}.

If $A \neq C_j$, then $A\eta_{k-i}$ is in M_{k-i}. So by induction hypothesis we have

$$A\eta_{k-i}\eta_{k-i+1} \cdots \eta_k \in T_{£}^{i}(\emptyset) \subset T_{£}^{i+1}(\emptyset)$$

since $T_{£}$ is monotonic.

On the other hand $A = C_j$. By induction hypothesis we have

$$B_q\theta\eta_{k-i+1} \cdots \eta_n \in T_{£}^{i}(\emptyset)$$

for all $1 \leq q \leq n$. So that

$$A\eta_{k-i}\eta_{k-i+1} \cdots \eta_k = B_0\theta\eta_{k-i}\eta_{k-i+1} \cdots \eta_k \in T_{£}^{i+1}(\emptyset)$$

by definition of $T_{£}$. (+)

Theorem 3 Let R be a recursive equational program and £ be a translated Horn program from E. Then

$$F(t_1, \ldots, t_n) \overset{*}{\underset{(p)}{\Rightarrow}}_R t_{n+1}$$

for all atoms $F_p(t_1, \ldots, t_{n+1})$ in lfp(£).

proof. We show by induction on $i \geq 1$ that

$$F(t_1, \ldots, t_n) \overset{*}{\underset{(p)}{\Rightarrow}}_R t_{n+1}$$

for all $F_P(t_1, \ldots, t_{n+1}) \in T_{£}^{i}(\emptyset)$.

If i = 1, then

$$F_P(t_1, \ldots, t_{n+1}) = F_P(q_1, \ldots, q_{n+1})\theta$$

for some definite sequent $F_p(q_1, \ldots, q_{n+1})$:- and for some substitution θ. This sequent corresponds to the term rewriting rule

$$F(q_1, \ldots, q_n) \text{ --> } q_{n+1}$$

by the transformation algorithm. So we have

$$F(t_1, \ldots, t_n) = F(q_1\theta, \ldots, q_n\theta)$$
$$(P)\Rightarrow_R \; t_{n+1}$$

by applying the rule $F(q_1, \ldots, q_n) \rightarrow q_{n+1}$ with the substitution θ. This is the induction basis.

Let $i \geq 1$. Suppose that

$$F(t_1, \ldots, t_n) \; (p)\overset{*}{\Rightarrow}_R \; t_{n+1}$$

for all atoms $F_P(t_1, \ldots, t_n, t_{n+1}) \in T_{\mathfrak{L}}^{i}(\emptyset)$. Let $F_P(t_1, \ldots, t_n, t_{n+1})$ be any atom in $T_{\mathfrak{L}}^{i+1}(\emptyset)$. By definition of $T_{\mathfrak{L}}$ there is a definite sequent $B_0 :- B_1, \ldots, B_m$ in $\mathfrak{L}$ such that

$$F_P(t_1, \ldots, t_n, t_{n+1}) = B_0\theta,$$
$$B_i\theta \in T_{\mathfrak{L}}^{i}(\emptyset), \; 1 \leq i \leq n$$

for some substitution θ. Let $F(q_1, \ldots, q_n) \rightarrow E$ be a rewriting rule from which definite sequent $B_0 :- B_1, \ldots, B_m$ is obtained. By applying rule $F(q_1, \ldots, q_n) \rightarrow E$ with the substitution θ to the $\sum$-term $F(t_1, \ldots, t_n)$, we can derive a $\sum$-term $E\theta$ as a result. To prove Theorem if suffices to show that for any subterm $G(E_1, \ldots, E_k)$ of $E\theta$, where G is the defined function symbol, if $G_P(p_1, \ldots, p_k, p_{k+1})$ is the corresponding atom which belongs to $(B_1, \ldots, B_m)\theta$, then

$$G(E_1, \ldots, E_k) \; (P)\overset{*}{\Rightarrow}_R \; p_{k+1}$$

follows from

$$E_j \; (P)\overset{*}{\Rightarrow}_R \; p_j, \; 1 \leq j \leq k.$$

This can be easily verified by using induction hypothesis. So we omit details. (+)

We obtain the following theorem for recursive equational programs from Corollary 4, Theorem 2, and Theorem 3.

Theorem 4 Let R be a recursive equational program and £ an translated Horn program from R. For any $\sum$-term $F(t_1, \ldots t_n)$ and for any constructor term t the next two conditions are equivalent.

(1) There is a successful computation of the input term $F(t_1, \ldots t_n)$ with the result t in the equational program R :

$$F(t_1, \ldots, t_n) \underset{(P)}{\overset{*}{\Rightarrow}}_R t.$$

(2) There is a successful computation of the goal $F_P(t_1, \ldots, t_n, t)$ in the logic program £ :

$$F_P(t_1, \ldots, t_n, t) \overset{*}{\Rightarrow}_{£} e. \qquad (+)$$

References

(1) K.R. Apt, M.H. Van Emden : Contributions to the theory of logic programming, J.ACM, 29, pp. 841-862 (1982).

(2) A. Arnold, M. Nivat : Formal computations of non deterministic recursive program schemes. Math. Systems Theory, 13, pp.219-236 (1980).

(3) C.L. Chang, R.C.T. Lee : Symbolic logic and mechanical theorem-proving, Academic Press New York (1973).

(4) P.J. Donway, R. Sethi : Correct computation rules for recursive language. SIAM J. Comput., 5, pp.378-401 (1976).

(5) M.H. Van Emden, R.A. Kowalski : The semantics of predicate logics as a programming language. J. ACM, 23, pp.733-742 (1976).

(6) J. Guttag, E. Horowitz, D.R. Musser : The design of data type specifications. in Current Trend in Programming Methodology, 4,

pp.60-79 (1978).

(7) L. Henschen, L. Wos : Unit refutations and Horn sets, J.ACM, 21, pp. 590-605 (1974).

(8) C.H. Hoffmann, M.J.O'Donnell : Programming with equations. ACM Trans. on Programming Languages and Systems, 4, pp.83-112 (1982).

(9) C.J. Hogger : Derivation of logic programs, J.ACM, 28, pp. 372-392 (1981).

(10) G. Huet : Confluent reductions : abstract properties and applications to term rewriting systems, J.ACM, 27, pp. 797-821 (1980).

(11) G. Huet, D.C. Oppen : Equations and rewrite rules. inFormal Language Theory, Ed. R.V. Book, Academic Press, pp.349-393 (1980).

(12) R. A. Kowalski : Algorithm + Logic + Control, C.ACM, 22, pp. 424-436 (1979).

(13) Z. Manna : Mathematical theory of computation, McGrqw-Hill (1974).

(14) M.J. O'Donnell : Computing in systems described by equations, Lec. Notes in Comput. Sci., NO 58 (1977).

(15) B.K. Rosen : Tree-manipulating systems and Church-Rosser theorems, J.ACM, 20, pp. 160-187 (1973).

(16) G.J. VanderBrug, J. Minker : State-Space, Problem-reduction, and theorem proving - Some relationship. C.ACM, 18, pp.107-115 (1975).